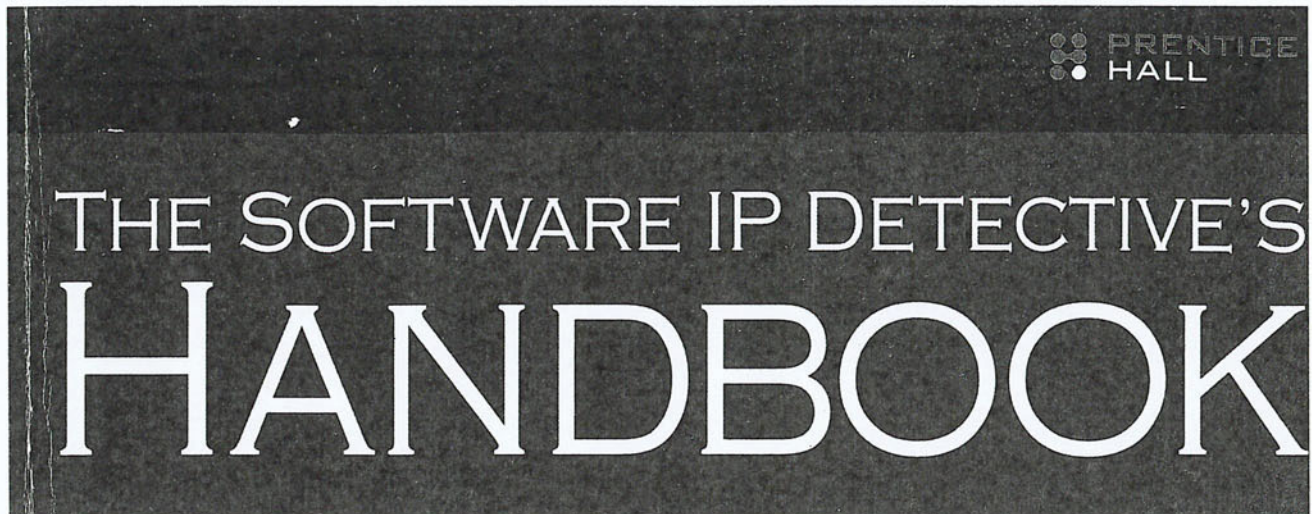
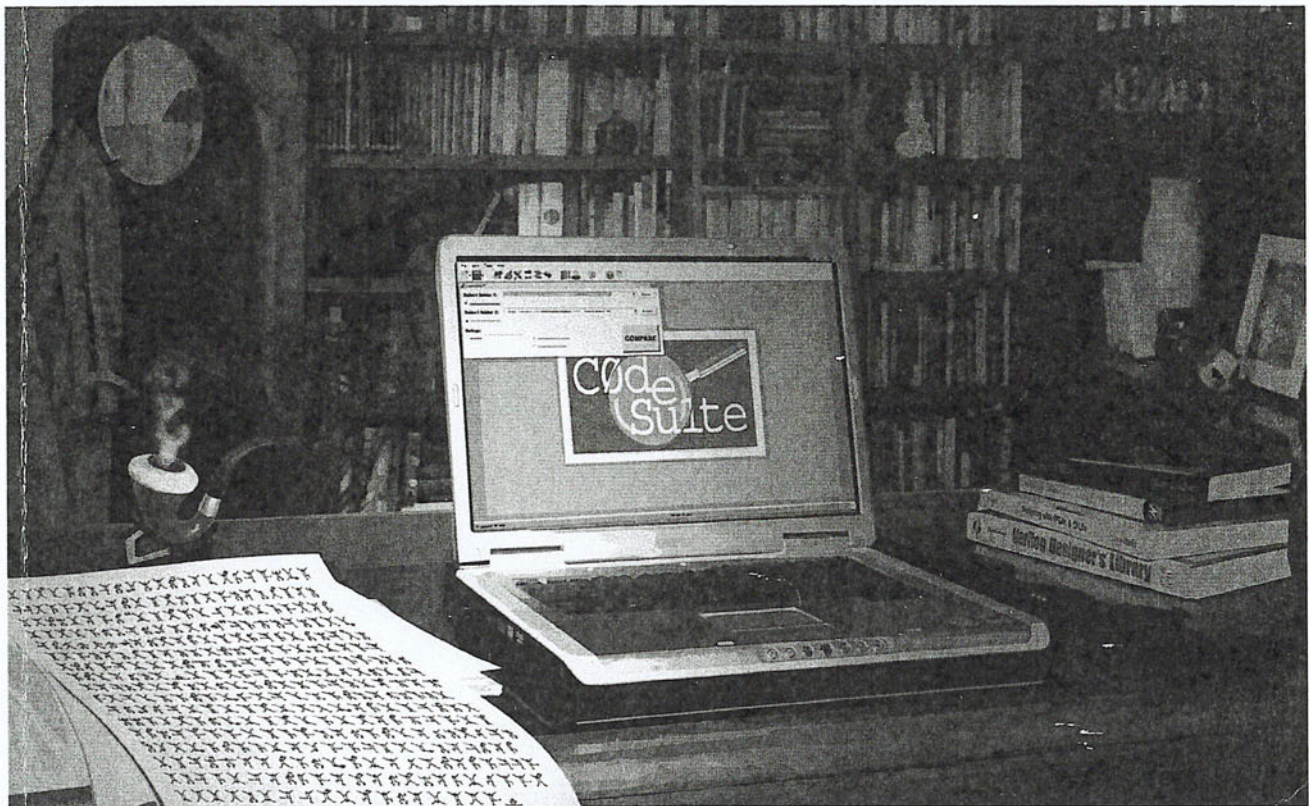


Exhibit A



Measurement, Comparison, and Infringement Detection



BOB ZEIDMAN

EXHIBIT 5
WIT: ZEIDMAN
DATE: 9/28/18
REPORTER: J. HARMONSON

THE SOFTWARE IP DETECTIVE'S HANDBOOK

The Software IP Detective's Handbook

**MEASUREMENT, COMPARISON, AND
INFRINGEMENT DETECTION**

Robert Zeidman

Software Analysis and Forensic Engineering Corporation



**PRENTICE
HALL**

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data is on file with the Library of Congress

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-13-703533-5
ISBN-10: 0-13-703533-0

This product is printed digitally on demand.

First printing, April 2011

Editor-in-Chief

Mark L. Taub

Acquisitions Editor

Bernard Goodwin

Managing Editor

John Fuller

Full-Service

Production Manager

Julie B. Nahil

Copy Editor

Barbara Wood

Indexer

Lenity Mauhar

Proofreader

Linda Begley

Cover Designer

Anne Jones

Compositor

LaurelTech

CONTENTS

Preface	xxi
Acknowledgments	xxiii
About the Author	xxv
PART I	
INTRODUCTION	I
Objectives	2
Intended Audience	2
Chapter I	5
About this Book	5
Part I: Introduction	6
Part II: Software	6
Part III: Intellectual Property	6
Part IV: Source Code Differentiation	9
Part V: Source Code Correlation	9
Part VI: Object and Source/Object Code Correlation	10
Part VII: Source Code Cross-Correlation	10
Part VIII: Detecting Software IP Theft and Infringement	11
Part IX: Miscellaneous Topics	11
Part X: Past, Present, and Future	11
Chapter 2	13
Intellectual Property Crime	13
2.1 The Extent of IP Theft	14
2.1.1 Unintentional Loss	14
2.1.2 Poor Economy	15
	ix

x CONTENTS

	2.1.3 Cheap Labor	16
	2.1.4 Criminal Operations	17
PART II	SOFTWARE	21
	Objectives	22
	Intended Audience	22
Chapter 3	Source Code	23
	3.1 Programming Languages	24
	3.2 Functions, Methods, Procedures, Routines, and Subroutines	26
	3.3 Files	32
	3.4 Programs	35
	3.5 Executing Source Code	36
	3.5.1 Compilers	36
	3.5.2 Interpreters	36
	3.5.3 Virtual Machines	36
	3.5.4 Synthesizers	37
Chapter 4	Object Code and Assembly Code	39
	4.1 Object Code	39
	4.2 Assembly Code	40
	4.3 Files	43
	4.4 Programs	44
Chapter 5	Scripts, Intermediate Code, Macros, and Synthesis Primitives	45
	5.1 Scripts	45
	5.2 Intermediate Code	47
	5.3 Macros	48
	5.4 Synthesis Primitives	49
PART III	INTELLECTUAL PROPERTY	53
	Objectives	55
	Intended Audience	55
Chapter 6	Copyrights	57
	6.1 The History of Copyrights	57
	6.2 Copyright Protections	60
	6.3 Software Copyrights	63
	6.3.1 Copyrighting Code	64
	6.3.2 Copyrighting Screen Displays	67

CONTENTS xi

	6.3.3	Registering a Software Copyright	70
	6.3.4	Term of Copyright Protection	71
	6.4	Allowable and Nonallowable Uses of Copyrighted Code	72
	6.4.1	Fair Use	72
	6.4.2	Reimplementation	72
	6.4.3	Versions, Modifications, and Derivatives	73
	6.4.4	Compilations	73
	6.4.5	Reverse Engineering	74
Chapter 7		Patents	79
	7.1	The History of Patents	80
	7.2	Types of Patents	81
	7.3	Parts of a Patent	82
	7.4	Patenting an Invention	85
	7.5	Special Types of Patent Applications	86
	7.5.1	Provisional	86
	7.5.2	Divisional	87
	7.5.3	Continuation	88
	7.5.4	Continuation-in-Part (CIP)	88
	7.6	Software Patents	90
	7.7	Software Patent Controversy	91
	7.7.1	Arguments for Software Patents	91
	7.7.2	Arguments against Software Patents	92
	7.7.3	The Supreme Court Rules?	94
	7.8	Patent Infringement	95
	7.8.1	Direct Infringement	95
	7.8.2	Contributory Infringement	96
	7.8.3	Induced Infringement	96
	7.8.4	Divided Infringement	97
	7.9	NPEs and Trolls	99
Chapter 8		Trade Secrets	103
	8.1	The History of Trade Secrets	103
	8.2	Uniform Trade Secrets Act (UTSA)	104
	8.3	Economic Espionage Act	105
	8.4	Aspects of a Trade Secret	106
	8.4.1	Not Generally Known	108
	8.4.2	Economic Benefit	108
	8.4.3	Secrecy	110

xii CONTENTS

	8.5 Trade Secret Theft	111
	8.6 Patent or Trade Secret?	112
Chapter 9	Software Forensics	113
	9.1 Forensic Science	115
	9.2 Forensic Engineering	116
	9.3 Digital Forensics	119
	9.4 Software Forensics	120
	9.5 Thoughts on Requirements for Testifying	121
	9.5.1 Certification	122
	9.5.2 Neutral Experts	123
	9.5.3 Testing of Tools and Techniques	124
PART IV	SOURCE CODE DIFFERENTIATION	125
	Objectives	126
	Intended Audience	126
Chapter 10	Theory	127
	10.1 <i>Diff</i>	128
	10.1.1 <i>Diff</i> Theory	128
	10.1.2 <i>Diff</i> Implementation	131
	10.1.3 False Positives	131
	10.1.3 False Negatives	132
	10.2 Differentiation	133
	10.2.1 Definitions	134
	10.2.2 Axioms	134
	10.3 Types of Similarity	135
	10.3.1 Mutual Similarity	135
	10.3.2 Directional Similarity	136
	10.4 Measuring Similar Lines	136
	10.4.1 Simple Matching	137
	10.4.2 Fractional Matching	137
	10.4.3 Weighted Fractional Matching	139
	10.4.4 Case Insensitivity	140
	10.4.5 Whitespace Reduction	140
	10.5 Measuring File Similarity	140
	10.6 Measuring Similar Programs	142
	10.6.1 Maximizing Based on Files	143
	10.6.2 Maximizing Based on Programs	144

CONTENTS xiii

Chapter 11	Implementation	147
	11.1 Creating and Comparing Arrays	147
	11.1.1 Sorting Lines	149
	11.1.2 Ordering Lines	149
	11.1.3 Hashing Lines	150
	11.1.4 Saving Arrays to Disk	151
	11.2 Number of Optimal Match Score Combinations	151
	11.3 Choosing Optimal Match Scores for Calculating File Similarity	153
	11.3.1 Simple Matching	153
	11.3.2 Fractional Matching	156
	11.4 Choosing File Similarity Scores for Reporting Program Similarity	161
	11.4.1 Choosing File Pairs for Optimally Determining Program Similarity	161
	11.4.2 Choosing File Pairs for Approximately Determining Program Similarity	162
Chapter 12	Applications	165
	12.1 Finding Similar Code	165
	12.1.1 Comparing Multiple Versions of Potentially Copied Code	165
	12.1.2 Use of Third-Party Code	167
	12.1.3 Use of Open Source Code	168
	12.2 Measuring Source Code Evolution	168
	12.2.1 Lines of Code	169
	12.2.2 Halstead Measures	170
	12.2.3 Cyclomatic Complexity	171
	12.2.4 Function Point Analysis	172
	12.2.5 How Source Code Evolves	173
	12.2.6 Changing Lines of Code Measure (CLOC)	175
	12.2.7 Non-Header Files	181
PART V	SOURCE CODE CORRELATION	183
	Objectives	185
	Intended Audience	185
Chapter 13	Software Plagiarism Detection	187
	13.1 The History of Plagiarism Detection	187
	13.2 Problems with Previous Algorithms	189
	13.3 Requirements for Good Algorithms	192

xiv CONTENTS

Chapter 14	Source Code Characterization	197
	14.1 Statements	199
	14.1.1 Special Statements	201
	14.1.2 Instructions	202
	14.1.3 Identifiers	204
	14.2 Comments	206
	14.3 Strings	207
Chapter 15	Theory	209
	15.1 Practical Definition	210
	15.1.1 Statement Correlation	211
	15.1.2 Comment/String Correlation	211
	15.1.3 Identifier Correlation	213
	15.1.4 Instruction Sequence Correlation	213
	15.1.5 Overall Source Code Correlation	213
	15.2 Comparing Different Programming Languages	213
	15.3 Mathematical Definitions	214
	15.4 Source Code Correlation Mathematics	215
	15.4.1 Commutativity Axiom	215
	15.4.2 Identity Axiom	216
	15.4.3 Location Axiom	216
	15.4.4 Correlation Definition	216
	15.4.5 Lemma	216
	15.5 Source Code Examples	216
	15.6 Unique Elements	218
	15.7 Statement Correlation	219
	15.7.1 Statement Correlation Equations	219
	15.7.2 Calculating the Statement Correlation	220
	15.8 Comment/String Correlation	223
	15.8.1 Comment/String Correlation Equations	223
	15.8.2 Calculating the Comment/String Correlation	224
	15.9 Identifier Correlation	225
	15.10 Instruction Sequence Correlation	227
	15.11 Overall Correlation	228
	15.11.1 S-Correlation	228
	15.11.2 A-Correlation	229
	15.11.3 M-Correlation	229
	15.11.4 Recommended Correlation Method	230

CONTENTS xv

Chapter 16	Implementation	233
	16.1 Creating Arrays from Source Code	234
	16.2 Statement Correlation	239
	16.3 Comment/String Correlation	240
	16.4 Identifier Correlation	241
	16.5 Instruction Sequence Correlation	243
	16.6 Overall Correlation	245
	16.7 Comparing Programs in Different Programming Languages	246
	16.8 Comparing Sections of Code Other than Files	246
Chapter 17	Applications	247
	17.1 Functional Correlation	248
	17.2 Identifying Authorship	249
	17.3 Identifying Origin	251
	17.4 Detecting Copyright Infringement (Plagiarism)	252
	17.5 Detecting Trade Secret Theft	252
	17.6 Locating Third-Party Code (Open Source)	253
	17.7 Compiler Optimization	254
	17.8 Refactoring	254
	17.9 Detecting Patent Infringement	255
PART VI	OBJECT AND SOURCE/OBJECT CODE CORRELATION	257
	Objectives	258
	Intended Audience	259
Chapter 18	Theory	261
	18.1 Practical Definition	266
	18.2 Extracting Elements	268
	18.2.1 Comment/String Elements	269
	18.2.2 Identifier Elements	269
	18.3 Comparing Different Programming Languages	270
	18.4 Mathematical Definitions	270
	18.5 Object and Source/Object Code Correlation Mathematics	272
	18.5.1 Commutativity Axiom	272
	18.5.2 Identity Axiom	272
	18.5.3 Location Axiom	272
	18.5.4 Correlation Definition	272
	18.5.5 Lemma	273

xvi CONTENTS

	18.6	Comment/String Correlation	273
	18.6.1	Comment/String Correlation Equations	273
	18.7	Identifier Correlation	274
	18.8	Overall Correlation	275
	18.8.1	S-Correlation	276
	18.8.2	A-Correlation	276
	18.8.3	M-Correlation	276
	18.9	False Negatives	276
Chapter 19		Implementation	279
	19.1	Creating Text Substring Arrays from Object Code	279
	19.2	Creating Arrays from Source Code	283
	19.2.1	Extracting Identifiers	283
	19.2.2	Extracting Comments and Strings	284
	19.2.3	String Delimiters	284
	19.2.4	Escape Characters	284
	19.2.5	Substitution Characters	286
	19.3	Identifier Correlation	287
	19.4	Comment/String Correlation	287
	19.5	Overall Correlation	287
Chapter 20		Applications	289
	20.1	Pre-Litigation Detective Work	289
	20.1.1	The Code in Dispute Is Distributed as Object Code	290
	20.1.2	A Third Party Is Bringing the Suit	290
	20.1.3	The Original Source Code Cannot Be Located	292
	20.2	Tracking Malware	293
	20.3	Locating Third-Party Code	293
	20.4	Detecting Open Source Code License Violations	294
PART VII		SOURCE CODE CROSS-CORRELATION	295
		Objectives	296
		Intended Audience	296
Chapter 21		Theory, Implementation, and Applications	299
	21.1	Comparing Different Programming Languages	300
	21.2	Mathematical Definitions	300
	21.3	Source Code Cross-Correlation Mathematics	301
	21.3.1	Commutativity Axiom	301
	21.3.2	Identity Axiom	302

CONTENTS xvii

	21.3.3 Location Axiom	302
	21.3.4 Correlation Definition	302
	21.3.5 Lemmas	303
	21.4 Source Code Examples	303
	21.5 Statement-to-Comment/String Correlation	307
	21.6 Comment/String-to-Statement Correlation	308
	21.7 Overall Correlation	308
	21.7.1 S-Correlation	309
	21.7.2 A-Correlation	309
	21.7.3 M-Correlation	309
	21.8 Implementation and Applications	310
PART VIII	DETECTING SOFTWARE IP THEFT AND INFRINGEMENT	313
	Objectives	315
	Intended Audience	315
Chapter 22	Detecting Copyright Infringement	317
	22.1 Reasons for Correlation	318
	22.1.1 Third-Party Source Code	318
	22.1.2 Code Generation Tools	319
	22.1.3 Commonly Used Elements	321
	22.1.4 Commonly Used Algorithms	322
	22.1.5 Common Author	325
	22.1.6 Copying (Plagiarism, Copyright Infringement)	325
	22.2 Steps to Find Correlation Due to Copying	326
	22.2.1 Filtering Out Correlation Due to Third-Party Code	326
	22.2.2 Filtering Out Correlation Due to Automatically Generated Code	327
	22.2.3 Filtering Out Correlation Due to Common Elements	328
	22.2.4 Filtering Out Correlation Due to Common Algorithms	329
	22.2.5 Filtering Out Correlation Due to Common Author	329
	22.2.6 Any Correlation Remaining Is Due to Copying	330
	22.3 Abstraction Filtration Comparison Test	331
	22.3.1 Background	331
	22.3.2 How CodeSuite Implements the Test	333
	22.3.3 Problems with the Test	334
	22.4 Copyright Infringement Checklist	338

xyiii CONTENTS

Chapter 23	Detecting Patent Infringement	341
	23.1 Interpreting the Claims	341
	23.1.1 Markman Hearing	345
	23.1.2 The Role of Experts at a Markman Hearing	347
	23.2 Examining the Software	348
	23.2.1 Searching for Comments	348
	23.2.2 Searching for Identifier Names	349
	23.2.3 Reviewing from a High Level	349
	23.2.4 Instrumenting Running Software	349
	23.3 Tools	350
	23.3.1 Understand	350
	23.3.2 Klocwork Insight	351
	23.3.3 DMS Software Reengineering Toolkit	351
	23.3.4 Structure101	352
	23.4 Determining Patent Validity	352
	23.4.1 Invalidity Based on MPEP 35 U.S.C. § 102	353
	23.4.2 Invalidity Based on MPEP 35 U.S.C. § 103	354
	23.4.3 Invalidity Based on MPEP 35 U.S.C. § 112	356
Chapter 24	Detecting Trade Secret Theft	359
	24.1 Identifying Trade Secrets	360
	24.1.1 Top-Down versus Bottom-Up	360
	24.1.2 Input from Owner	361
	24.1.3 State with Specificity	361
	24.1.4 Reasonable Efforts to Maintain Secrecy	364
	24.1.5 Copied Code as Trade Secrets	365
	24.1.6 Public Sources	366
	24.2 Tools	367
PART IX	MISCELLANEOUS TOPICS	369
	Objectives	370
	Intended Audience	370
Chapter 25	Implementing a Software Clean Room	371
	25.1 Background	372
	25.2 The Setup	374
	25.2.1 The Dirty Room	374
	25.2.2 The Clean Room	375
	25.2.3 The Monitor	376

CONTENTS xix

	25.3 The Procedure	376
	25.3.1 Write a Detailed Description	377
	25.3.2 Sign Agreements	379
	25.3.3 Examine the Original Code in the Dirty Room	380
	25.3.4 Transfer Specification to the Monitor and Then to the Clean Room	380
	25.3.5 Begin Development in the Clean Room	380
	25.3.6 Transfer Questions to the Monitor and Then to the Dirty Room	380
	25.3.7 Transfer Answers to the Monitor and Then to the Clean Room	381
	25.3.8 Continue Process Until New Software Is Completed	381
	25.3.9 Check Final Software for Inclusion of Protected IP	381
	25.3.10 Violations of the Procedure	381
Chapter 26	Open Source Software	383
	26.1 Definition	383
	26.2 Free Software	386
	26.2.1 Copyleft	387
	26.2.2 Creative Commons	387
	26.2.3 Patent Rights	387
	26.3 Open Source Licenses	388
	26.4 Open Source Lawsuits	390
	26.4.1 <i>SCO v. Linux</i>	391
	26.4.2 The BusyBox Lawsuits	393
	26.4.3 <i>IP Innovation LLC v. Red Hat and Novell</i>	394
	26.4.4 <i>Network Appliance, Inc. v. Sun Microsystems, Inc.</i>	395
	26.5 The Pervasiveness of Open Source Software	396
Chapter 27	Digital Millennium Copyright Act	399
	27.1 What Is the DMCA?	399
	27.2 For and Against the DMCA	400
	27.3 Noteworthy Lawsuits	403
	27.3.1 <i>Adobe Systems Inc. v. Elcom Ltd. and Dmitry Sklyarov</i>	403
	27.3.2 <i>MPAA v. RealNetworks Inc.</i>	404
	27.3.3 <i>Viacom Inc. v. YouTube, Google Inc.</i>	404

xx CONTENTS

PART X	CONCLUSION: PAST, PRESENT, AND FUTURE	407
Glossary		409
References		423
Index		435

16 IMPLEMENTATION

Now that I have discussed the theory behind source code correlation, I will discuss ways of practically and efficiently implementing that theory.¹ The particular implementation discussed in this chapter is based on the implementation used in the commercial CodeMatch tool that is a function of the CodeSuite program available from S.A.F.E. Corporation. This tool focuses on finding software copyright infringement, so many of the implementation choices are based on optimizing that use of source code correlation.

The implementation I describe here makes use of a basic knowledge of programming languages and program structures to simplify the task of comparing and identifying matching program elements. It is not necessary to implement a full programming language parser because the exact functionality of the statements does not need to be known (this will not be the case, however, when functional correlation is developed). The implementation must only recognize programming language keywords that are specific to the programming language of the program being examined. In addition, this implementation needs to know the characters that are used to delimit comments, the characters that are used to delimit strings, and the characters called “separators,” which are the characters that cannot be used in identifier names. For example, operators (such as * / + -) are separators because when parsing a statement, reaching one of these means

1. Note that some of the concepts discussed in this chapter may be covered by patents that have been issued or are pending.

234 IMPLEMENTATION

the end of an identifier has been reached (typically, though not always, whitespace is reached first). On the other hand, the underscore character (`_`) is valid within an identifier name in many programming languages, so it is not a separator.

The implementation needs to separate programs into five elements: statements, comments, strings, instructions, and identifiers. Note that these are not mutually exclusive; statements can include instructions, identifiers, and strings. It is useful to put these elements of code into arrays.

16.1 CREATING ARRAYS FROM SOURCE CODE

The first step in creating the arrays is to create two arrays for each source code file being examined. Consider the small snippet of C code shown in Listing 16.1.

Listing 16.1 Code Snippet I

```
// ---- begin routine ----
void fdiv(
    char *fname,          // file name
    char *pathString)     // path
{
    int Index1, j;
    printf("Hello world!\n");

    if (strlen(pathString) == 0)
        strcpy(pathString, "C:\\Windows\\");
    else
        strcat(pathString, "\\");

    while (Index1 < 100)
    {
        printf("Getting results.\n");
        j = strlen(fname);
        j += strlen(PathString);
        /* find the file extension */
    }
}
```

Table 16.1 shows how this small sample would lead to two arrays of statements and comments/strings. Note that whitespace has been reduced. This involves trimming all whitespace characters on the right and left of each string and

17

APPLICATIONS

There are many potential applications for source code correlation. In this chapter I discuss some of the ones that I have identified, though there are likely many more than these. Table 17.1 shows the applications that I discuss in this chapter and the most useful element correlations to emphasize in each case; each application is likely to emphasize certain element correlations over others, though these are not hard-and-fast rules.

Table 17.1 Applications of Source Code Correlation

Type of application	Emphasized correlations
Identifying authorship	Comment/string correlation Identifier correlation
Identifying origin	Comment/string correlation Identifier correlation
Detecting copyright infringement (plagiarism)	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation
Detecting trade secret theft	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation Functional correlation

Continues

248 APPLICATIONS

Table 17.1 Applications of Source Code Correlation (*Continued*)

Type of application	Emphasized correlations
Locating third-party (open source) code	Statement correlation Comment/string correlation Identifier correlation Instruction sequence correlation
Compiler optimization	Statement correlation Instruction sequence correlation Functional correlation
Refactoring	Statement correlation Comment correlation Instruction sequence correlation Functional correlation
Detecting patent infringement	Functional correlation

17.1 FUNCTIONAL CORRELATION

Note that Table 17.1 mentions “functional correlation.” This correlation involves functional elements of the code. I do not discuss the particulars of functional correlation in the book because it is a wide-open topic. Many techniques have been developed to define and analyze the functionality of programs. As of yet, I have not come across a tool that meets the following four criteria:

1. Is completely automatic
2. Is programming-language-independent
3. Completes in a reasonable amount of time
4. Produces scores that can fit into the source code correlation framework

The first criterion means that source code files are compared without human intervention. Once the tool is pointed at two sets of source code files, it calculates functional correlation without the need for the user to intervene. Certainly it is possible for a useful tool to calculate functional correlation and still require human intervention, but ideally such operator interference would not be necessary.

The second criterion means that two programs in two different programming languages could be compared. This is particularly important because functional

correlation should be comparing program functionality independently of the programming language. A starting point could be a tool that requires the programs being compared to be written in the same programming language. Such a tool could be useful but would be limited. Practical uses of functional correlation such as detecting trade secret theft and patent infringement would be most useful if one were able to compare programs written in different programming languages.

The third criterion has been a difficult goal to reach so far. Analyzing the functionality of a program seems to take exponential time with respect to the size of the program. Comparing two programs involves creating an abstract representation of the functionality of each program (something not well defined at this time and for which there is currently no standard) and comparing the two abstract representations. Many of the efforts to date attempt to create a representation of the entire program, which certainly makes a lot of sense. However, given the computational time constraints, perhaps it makes more sense to break the functionality down file by file and simply compare files as is done for the other correlation elements (statements, comments/strings, identifiers, and instruction sequences).

Once functionality can be automatically extracted from a program in a reasonable amount of time and independently of the programming language, the final criterion is to compare two functional representations in such a way that a normalized score is created. It is an open question how to accomplish this.

17.2 IDENTIFYING AUTHORSHIP

To use source code correlation to determine authorship of a program or an algorithm within a program, it would be most useful to find correlation between those elements that would be most likely to be unique to particular programmers. These elements are the comments, strings, and identifier names. Obviously any correlation used to determine authorship would emphasize comment/string correlation and identifier correlation.

There are instances when it is important to identify the author of source code. This could occur during legal disputes when two or more parties claim to have written a particular program. It could occur when two parties have jointly developed code for a program and, because of a contractual dispute or a royalty determination, must figure out which code was written by which party. While this may seem like an unusual scenario, I have been personally involved in several

250 APPLICATIONS

cases where a large software manufacturer has asked me to search for its own programs online, in surplus software stores, and in my own private computer software collection because the company cannot find an earlier version of its software. This occurs more often than you might think in the software industry, sometimes because of the lack of strict procedures at many companies. It also happens because many modern software companies start very small and grow very large in a short period of time, and the emphasis is on getting products out faster than their larger competitors. The limited resources and short market windows mean that developing software quickly takes precedence over installing and using version control systems to keep track of software changes.

Determining authorship can also be useful in tax cases. Tax rates vary according to jurisdictions (for example, city, county, state, country). When software is partially developed overseas, for example, the IRS of the U.S. government can tax the foreign-developed code at a different rate from the domestically developed code. When multiple versions of code have been developed in multiple countries by multiple teams using various tools, the company may not have kept track of which code was developed where. Again, source code correlation can be used to help make this determination.

Identifying authorship can be useful to computer historians to determine whether certain programmers are responsible for writing certain programs or portions of programs. Historians use this kind of technique to determine authorship of works of literature and works of art. As computers become a part of history, computer historians may need to do something similar to correctly attribute code to its authors. Literary works are almost always published with the author's name on the work, yet there are still controversies and there are known cases of fraud. There are those who believe that William Shakespeare was the pseudonym for writer Christopher Marlowe, philosopher and scientist Francis Bacon, or Edward de Vere, the 17th Earl of Oxford and Lord Great Chamberlain of England. There have also been famous forgeries of artwork. One such example is three Etruscan terra-cotta warriors. They were purchased in the years 1915 through 1918 and put on display in 1933 as examples of the work of the ancient Etruscans, a people that lived in Italy around 800 BCE. It was not until 1960 that chemical tests determined that modern materials were used and that, in fact, the pieces were created at the time of their sale by Italian brothers Pio and Alfonso Riccardi and their sons.

While it seems unlikely that source code would ever be as valuable as a historical artifact to warrant the attention of forgers, historians have already begun collecting, studying, and storing computer hardware and software in places like

the Computer History Museum in Mountain View, California. It is conceivable that these historians may want to determine or even locate the programmer for a particular piece of code, and source code correlation can help.

Another use for identifying authorship is for tracking viruses, spyware, and other kinds of cyber-security threats. Identifying the origin of software, as described in the next section, as well as authorship is useful in this important field. Comparing source code for a computer virus against source code from known programmers, particularly those with a history of such illicit activity, can help catch these criminals and convict them at trial.

17.3 IDENTIFYING ORIGIN

As with determining authorship of a program or an algorithm within a program, identifying origin focuses on the correlation between those elements that would most likely be unique to programmers in particular companies or countries. These elements are the comments, strings, and identifier names. Any correlation used to determine origin would emphasize comment/string correlation and identifier correlation.

Identifying the origin of a program or a piece of a program can be useful for the same reasons as identifying authorship is useful. Because many modern programs are written by teams of engineers, it may not be possible to identify the actual author, but it may be possible to identify the company where the software was developed. Different companies have different coding standards and different corporate cultures that could result in higher correlation between programs developed within the company than between programs developed at different companies.

Similarly, it may be possible to identify the country of origin of a program. In particular, not only might coding standards and coding styles be similar among programmers in a particular country, but there would obviously be language differences in the identifier names and particularly in the comments and strings. Even in countries where programmers use the same language, idioms, colloquialisms, and slang would differentiate countries or areas within countries where the software was developed.

Identifying the country of origin can be useful in legal disputes, as described previously. Country of origin may be even more useful in tracing software intended to infiltrate another country's in an act of cyber warfare, a growing and dangerous threat in the modern world that can potentially bring down an entire nation without a single missile or weapon of mass destruction.

17.4 DETECTING COPYRIGHT INFRINGEMENT (PLAGIARISM)

The most common use of source code correlation to date is to find copyright infringement, which is essentially the legal term for plagiarism. Source code correlation was initially designed to help detect copyright infringement, and it has been successfully used in over 50 cases at the time of this writing.

To find copyright infringement, it is necessary to find all signs of copying, especially considering that modifications may have been made subsequent to the unauthorized copying. The modifications may be explained by the normal development process, including debugging and feature enhancements and additions. The modifications may also be the result of deliberate attempts to hide the copying. For these reasons, source code correlation used to detect copyright infringement must emphasize correlation of all literal elements: statements, comments and strings, identifiers, and instruction sequences.

17.5 DETECTING TRADE SECRET THEFT

Another common use of source code correlation is the detection of trade secret theft. When literal elements of a program's source code are copied, not only does that constitute copyright infringement, but it is also often a case of trade secret theft. This is because much software source code, other than open source software, meets the requirements of being a trade secret that were discussed in Chapter 8:

1. It is not generally known to the public. Programmers may know how to implement particular functions using source code, but each program has many unique functions that are implemented in ways that are not generally known because they are specific to the particular program.
2. It confers some sort of economic benefit on its holder, where the benefit is from not being known to the public. Most software is difficult to write, taking years of education and experience to do correctly. Most code has an economic benefit to its owner because of the knowledge and effort that went into creating it. Also, it can be argued that understanding source code allows a competitor to learn a program's weaknesses that can be exploited, lowering the program's value.
3. The owner of the trade secret makes reasonable efforts to maintain its secrecy. Software source code is almost always kept secret.

Consequently, in the case of literal copying, not only is source code correlation useful for detecting copyright infringement, but it is also useful for finding

17.6 LOCATING THIRD-PARTY CODE (OPEN SOURCE) 253

trade secret theft using all four algorithms—statement correlation, comment/string correlation, identifier correlation, and instruction sequence correlation.

However, trade secret theft can also involve copying a method for accomplishing some function without copying the actual source code that implements that function. For this reason, source code correlation that would find all forms of trade secret theft would include functional correlation. As of today, there is not an efficient method for determining functional correlation in a reasonable amount of time. A tool for computing a correlation that can accurately detect trade secret theft is still in the future, as of this writing.

17.6 LOCATING THIRD-PARTY CODE (OPEN SOURCE)

As open source code is finding its way into more and more products, it is becoming a more complex issue. Open source code is covered by different licenses, the most common one being the GNU Public License (GPL) that has certain strict requirements, particularly about making source code available to the public. In many cases, open source code has crept into software projects without the managers or many of the programmers being aware. Sometimes open source code has been used in a project even though the project owners do not intend to abide by the licensing terms. Companies are now getting nervous about this phenomenon, especially because several successful lawsuits have been brought by the open source community of developers against commercial companies. Many companies now employ sophisticated means of determining whether their own software products contain open source code, and source code correlation can be used to help make that determination.

The issues associated with detecting open source code are really a subset of the issues associated with finding any third-party source code in a project. There may be instances of commercial third-party code for which the license has expired or that was obtained by unlawful means. Source code correlation can be a useful tool for detecting the use of such third-party code, whether it is open source or commercial.

In most cases, the third-party code has not been modified, or has not been modified significantly, because typically the source code is used unintentionally and so there is no attempt to obscure its use in the program. Even when the source code is used with the knowledge that it is not lawful, the users do not expect it to be uncovered because they are not releasing their source code (see Part VI). Also, third-party code is often not modified, or modified only slightly, because an advantage of third-party code is that it has been written

254 APPLICATIONS

and vigorously tested, and modifying the code would require further testing and debugging. In these cases, source code correlation may be overkill, and techniques that look for exact matches, like source code differentiation (see Part IV) or comparing hashes of the source code, may be sufficient.

However, if an attempt has been made to hide the third-party code, or it is believed that such an attempt was made, then source code correlation used to detect third-party code must emphasize the same elements as those used for detecting plagiarism. This involves all literal elements: statements, comments and strings, identifiers, and instruction sequences.

17.7 COMPILER OPTIMIZATION

It is possible that source code correlation can be used to eliminate redundancy in code during the compilation process, thereby optimizing the compilation. In this case, it seems that the correlation should emphasize statement correlation and instruction sequence correlation and, of course, functional correlation when that becomes available. By using source code correlation, a compiler could find highly correlated sections of code. Once a compilation has been completed for one of these sections of code, the compiler may be able to speed up the effort on highly correlated sections of code without needing to perform another full, time-consuming optimization process.

17.8 REFACTORING

An area of growing interest in computer science, and in industry, is refactoring. This is the process of cleaning up and simplifying software source code to make it more readable and maintainable, without changing its functionality. According to Martin Fowler, one of the early gurus of refactoring, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

One particular aspect of refactoring is clone detection and elimination. A clone is a copy of a section of code that performs a function that is identical to that of the original code. In an early paper on clone detection entitled "Using Clone Detection to Manage a Product Line," Ira Baxter and Dale Churchett describe clone detection and its benefits as follows:

17.9 DETECTING PATENT INFRINGEMENT 255

Clone detection finds code in large software systems that has been replicated and modified by hand. Remarkably, clone detection works because people copy conceptually identifiable blocks of code, and make only a few changes, which means the same syntax is detectably repeated. . . . Clones can thus enhance a product line development in a number of ways: removal of redundant code, lowering maintenance costs, identification of domain concepts for use in the present system or the next, and identification of parameterized reusable implementations.

Obviously it is unnecessary and inefficient to have cloned code within a single product because it just increases the code size and increases the maintenance effort without providing significant additional functionality. Eliminating clones is a useful exercise if it can be done efficiently, and source code correlation can be an efficient method of doing so. Correlation used for clone detection and elimination would emphasize the following elements of source code correlation: statement correlation, comment correlation, instruction sequence correlation, and functional correlation.

Notice that I refer to comment correlation rather than comment/string correlation. It might not seem obvious at first to include comment correlation. Here we are concerned mostly about the descriptions provided in the comments, and this may be one area where comments and strings should be treated differently rather than combined into a single correlation score. Comments, particularly headers that describe the functionality of a routine in the program, could provide important details about the correlation of two functions in the program. For example, it is possible that two functions, written by two different programmers, would differ significantly in their implementation but in fact implement the same function. Depending on the efficiency, and availability, of a functional correlation measure, comment correlation could be a way of determining that two or more functions perform the same operation and can be replaced with a single function, even though their other correlation scores are low.

17.9 DETECTING PATENT INFRINGEMENT

Patent infringement essentially involves the creation of a program that uses a patented method by someone other than the person or entity that owns the patent and has not been given that right by the patent owner. Patent infringement occurs even when the infringer has never seen the source code of the patented invention. For this reason, calculating the correlation of the literal

256 APPLICATIONS

elements of a program is generally not useful unless source code was also copied. For the vast majority of patent cases, source code correlation would emphasize functional correlation, which, as explained above, is not yet available. When an efficient, accurate method of functional correlation is found, there will be many uses for it.